# Creating other types of synchronization objects that can be used with co_await, part 8: The shared mutex

devblogs.microsoft.com/oldnewthing/20210318-00

March 18, 2021

Raymond Chen

Our next stop in showing off our library for building awaitable synchronization objects is the shared mutex. This one takes advantage of the “extra data” we had been hiding in our nodes.

```
template<>
struct async_helpers::
    extra_await_data<awaitable_shared_mutex_state>
{
    extra_await_data(bool kind) : exclusive(kind) {}
    bool exclusive;
};
```

We have some extra state to pass into the `claim` function, namely whether we are attempting an exclusive or shared claim. We specialize the `extra_await_data` structure to let the library know about it.

```
struct awaitable_shared_mutex_state :
    async_helpers::awaitable_state<awaitable_shared_mutex_state>
{
    std::atomic<int> owners;
```

We start by defining our state. I’ve decided to represent the three categories of states as follows: Owned exclusively (−1), not owned (0), and owned shared (positive shared owner count).

```
    static bool exclusive_transition(
        int current, int& future) noexcept
    {
        if (current != 0) return false;
        future = -1;
        return true;
    }

    static bool shared_transition(
        int current, int& future) noexcept
    {
        if (current < 0) return false;
        future = current + 1;
        return true;
    }
```

To acquire the shared mutex exclusively, the mutex must currently be completely unowned (value zero), and after we're done, it will be owned exclusively by us, which is represented by a state value of −1.

To acquire the shared mutex in shared mode, the mutex must either be unowned or owned in shared mode (value zero or positive), and after we're done, it will be have a share count one greater than before.

We choose the transition function depending on what kind of claim we are making.

```
    bool fast_claim(extra_await_data const& e) noexcept
    {
        if (e.exclusive) {
            return calc_claim<true>(owners, exclusive_transition, 0);
        } else {
            return calc_claim<true>(owners, shared_transition);
        }
    }

    bool claim(extra_await_data const& e) noexcept
    {
        if (e.exclusive) {
            return calc_claim<false>(owners, exclusive_transition);
        } else {
            return calc_claim<false>(owners, shared_transition);
        }
    }
```

A fast claim in exclusive mode makes the optimistic assumption that the current state of the shared mutex is unowned, since that's the only case that will succeed. Shared mode can succeed with multiple initial states, so we default to using whatever the current state is. Slow claims always use the actual current state.

```
    void unlock_exclusive(node_list& list)
    {
        resume_a_bunch(list);
    }

    void unlock_shared(node_list& list)
    {
        if (owners.fetch_add(-1,
            std::memory_order_relaxed) == 0) {
            resume_a_bunch(list);
        }
    }
```

When an exclusive lock is released, the mutex becomes unowned, and we can look for coroutines to resume. When a shared lock is released, we decrement the shared waiter count, and only if the mutex becomes unowned do we look for coroutines to resume.

```
    void resume_a_bunch(node_list& list)
    {
        auto count = 0;
        auto peek = peek_head();
        if (!peek) {
            // nobody to release
        } else if (peek->exclusive) {
            resume_one(list);
            count = -1;
        } else {
            do {
                ++count;
                resume_one(list);
                peek = peek_head();
            } while (peek && !peek->exclusive);
        }
        owners.store(count, std::memory_order_relaxed);
    }
};
```

To look for coroutines to resume, we peek at the first coroutine on the wait list. If there isn't one, then leave the shared mutex unowned. Otherwise, grant ownership to that first waiting coroutine, and if it was a shared request, then accumulate the next shared waiters as well.

Finally, we hook up these operations to the main class:

```
struct awaitable_shared_mutex :
    async_helpers::awaitable_sync_object<
        awaitable_shared_mutex_state>
{
    void operator co_await() = delete;

    auto lock_shared() { return make_awaiter(false); }
    auto lock_exclusive() { return make_awaiter(true); }

    void unlock_exclusive()
    {
        action_impl(&state::unlock_exclusive);
    }

    void unlock_shared()
    {
        action_impl(&state::unlock_shared);
    }
};
```

We delete the ability to `co_await` the shared mutex directly. You have to await the `lock_shared()` or `lock_exclusive()` method in order to make it clear which type of lock you want.

Now that I've written this class, I can tell you that it's broken.

Observe that if the mutex is locked shared, then additional shared requests are granted immediately, even if there is an exclusive lock waiting. This causes shared locks to starve out exclusive locks. We could try to address this by forcing lock requests to suspend if the wait queue is nonempty.

Another problem is that if we release the last shared lock, leaving the lock unowned, we ask `resume_a_bunch` to resume a bunch of waiting coroutines. However, since the lock is unowned, another thread can sneak in and steal the lock. That other thread may have stolen the lock in a way that is inconsistent with the lock we are about to grant.

So let's try again, next time.

Raymond Chen

**Follow**